Compiler Construction Project
CS 444

Peter Mielke 84xxxxxx
Markus Wandel 85020317

1. User's Guide

The compiler comes as a single executable file, named tcc. The easiest way to invoke the compiler is with the following command:

```
tcc file.c
```

This will read ''file.c'' run it through the C preprocessor, compile it and, assuming there were no errors, leave the assembly language output in a file named ''file.s'' This is most easily assembled and linked using the front end for the system compiler, using the following command:

```
cc file.s
```

Alternatively, tcc can compile its standard input and send the assembly language output to its standard output; thus, it can be used in a pipeline. To accomplish this, no file name is given. The C preprocessor is not invoked in this case. If it is desired that tcc compile from its standard input to an output file, then the output file can be specified with the ''-o'' option; this also overrides the default output file name in the other case. Use of the C preprocessor can only be used when using a named file and can be specifically overridden with the ''-n'' option, and compiling to the standard output can be enforced with the ''-c'' option. Therefore, all combinations of standard and file input/output are possible. The C preprocessor must be invoked separately if any options need to be given to it.

Finally, tcc has an option to insert a flag into its output which gives a permanent record that the file has been compiled with tcc. This is the ''-i'' option. When it is used, a line of the following form is inserted into the output:

```
 .ascii  "@(#) tcc compiled: file.c@(#)"
```

When this is assembled and linked, the flag line can always be found by examining the object file with the SCCS ''what'' command. Finally, tcc has a ''-h'' option; this gives the following help message:

```
usage: tcc [-n] [-i] [-c] [-o filename] filename
        -n             don't use c preprocessor (only for files)
        -c             send output to standard out
        -i             add version type to output
        -o filename    place output into filename
```

2. Where to find tcc Online

The source code and executable for tcc, as well as as number of tests, have been made available on machine ''lion'' They should be publicly readable. The directory paths are as follows:

lion:~mwandel/tcc/tcc - Executable for the compiler
lion:~mwandel/tcc/src - Source code for the compiler
lion:~mwandel/tcc/test - Test cases

To test recompilation of the compiler with itself, one need only copy the entire tcc directory tree somewhere where one has write access, go into the tcc/src directory, remove the ''.s'' and ''.o'' files, and type ''make''

3. Operation of the Front End

3.1. Lexical analysis

The front end of tcc is a standard (f)lex generated scanner. It is invoked by the parser via the function yylex(), which returns a token or zero. Some tokens have further information associated with them. These are the following:

1. Tokens which indicate numbers (`INTEGER`, `UNSINT`, `LONGINT`, etc.) return the actual number in the global variable yylval.
2. Tokens which indicate character strings (`SINGLECHAR`, `STRING`) return the length of the string in yylval and leave the string itself accessible in the global variable stringbuf. Escape sequences are processed before the string is returned. The string can contain embedded nulls and is not null terminated.
3. Tokens which indicate identifiers collect the identifier string, null-terminate it, and make a privately allocated copy of it. A pointer to this copy is returned in yylval. The scanner examines the symbol table to find if the identifier is a user defined type, and returns `ID` or `TYPE_ID` as the token appropriately.

The scanner collects its input as it is read, so it always has a copy of the current input line. When an input line ends, it checks if the error routine has flagged an error on that line, and if this is the case, the line is printed with a pointer to the error position underneath.

3.2. Parsing

The parser for tcc is produced by yacc/bison from a straightforward ANSI C grammar. The parser builds a parse tree using a variety of parse tree construction functions, which are also the interface to the type checker. The simplest parse tree construction function is defined as follows:

```
struct treenode *make_plain(type,l1,l2,l3)
int type;
struct treenode *l1, *l2, *l3;
```

It takes pointers to up to three nodes, which will become children of the newly created node. It returns the pointer to the newly created node, which is set to have the given type. If the node has less than three children, null pointers must be provided for the extra child node fields. Some places in the grammar call other parse tree construction functions, which additionally do type checking and other special processing.

Each parse tree node contains the following information:

- the type of the node.
- the line number in the source code which the syntactic construct represented by the node was on.
- the data type of the object represented by the node; this is a pointer to a ‘‘type chain’’ which will be discussed later, or null if not appropriate (in the case of control statement nodes).
- a ‘‘format’’ field identifying the format of the rest of the node.
- a variety of format-dependent fields (see ‘‘nodes.h’’).

Error recovery in the grammar is done by a few strategically placed ‘‘error’’ productions. These are invoked when the input to the parser cannot be matched by any syntactic construct known to the it. An error is reported in these cases and parsing continues with incorrect parse tree segments suitably disposed of. Parse trees are built up to the level of a declaration or a function body, then they are processed by calls to the declaration processor or code generator, and disposed of.

3.3. Error Reporting

Error reporting in tcc is done by calling one of the error functions. These are err() and errl(). err() takes two integer arguments. The first is a flag indicating whether the call is for an error (1) or a warning (0). The second is a number used to index into the table of error messages. Further arguments are needed to fill in details in some error messages; these are provided in the same manner as for printf(). err() takes the error location to be the current point in the input text, and instructs the scanner to print out the line with a pointer to the error when it has finished reading it. errl() takes an additional argument, which is a tree node. It takes the line number from the tree node, and indicates that the error is ‘‘near’’ that line. err() and errl() count errors (but not warnings) and if any errors occur during a compile, the output file is deleted at the end. This does not work if the compiler is compiling to the standard output.

3.4. Symbol Table and Type Chains

The C language has a number of basic types (such as integer and char) and several ‘‘type constructors’’ which can be applied to a basic type to make it a complex one. These are, for example, ‘‘pointer to’’, ‘‘array of’’, and ‘‘structure containing’’. Tcc internally represents types as chains of type nodes, called type chains. All type chains end with a basic type node. Type chains can merge into other type chains, but in this case, a ‘‘type ref’’ node is inserted just before the merge point to store information specific to that type chain, such as whether the object represented is constant or volatile.

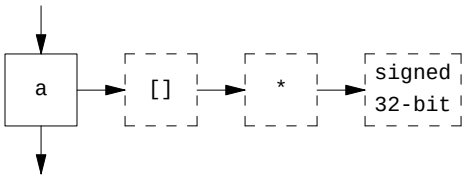
The symbol table implementation is very simple. The symbol table entries are kept in a singly linked list, and each entry contains, among other things, the name of the object it defines, a pointer to the type chain for the object, and a block level. The block level is zero for external objects, one for objects inside a function, and greater than one for objects in inner blocks of a function. The following functions provide the interface between the symbol table and the rest of the compiler:

```
void init_symtab()

struct symbol *make_sym(level,name,entrytype)
int level, entrytype;
char *name;

void delete_sym_level(level)
int level;

struct symbol *find_sym(name,entrytype,minlevel)
char *name;
int entrytype, minlevel;

struct symbol *find_next(ptr,name,entrytype,minlevel)
struct symbol *ptr;
char *name;
int entrytype, minlevel;
```

The first function initializes the symbol table to empty. The second function creates a symbol of the indicated level, name, and entry type, and links it into the symbol table in the proper place, that is, together with the other entries of the same block level. The third function deletes nodes in the symbol table until it comes to a node whose level is lower than the indicated one. The last two functions search for entries, given a name, a minimum block level, and an entry type. Entry types are bit masks, so a number of different entry types can be specified by logical OR of the entry types desired. Finally, the ‘‘name’’ argument can be left at null to indicate that the first appropriate symbol table entry should be returned.

The following diagrams show a few examples of symbol table entries and type chains. The first one would be produced by the following declaration:

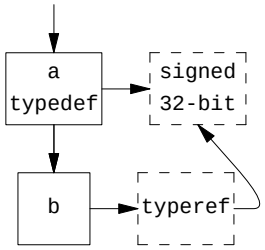
```
int *a[];
```



Here we see the symbol table entry for `a`, with an `array of` and a `pointer to` constructor applied to the basic type `signed 32-bit`, which is the tcc internal representation of `int`.

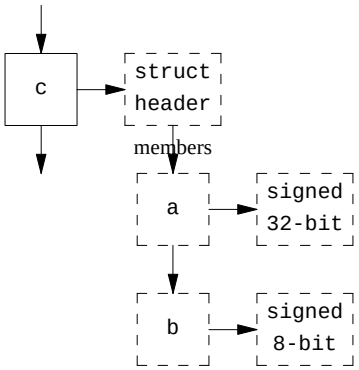
Next, the symbol table entry for a user defined type, and a use of that type, is shown:

```
typedef int a;
a b;
```

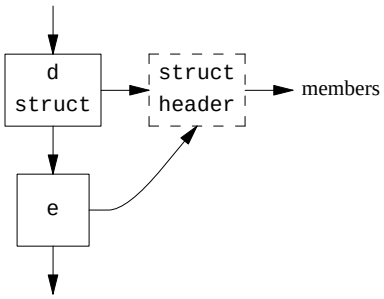


Here we see a special symbol table entry storing the type definition, and a second symbol table entry making use of the type definition via a `type ref` node.
Two examples of type chains for structure definitions follow next; the first one shows how the members of a structure are stored, the second one shows how a named structure is stored in the symbol table and how other symbol table entries can use it. In this case, `e` does not require a `type ref` node since the structure was defined together with `e`.

```
struct { int a; char b; } c;
```

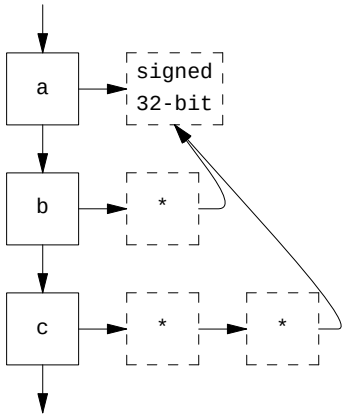


```
struct d { ... } e;
```



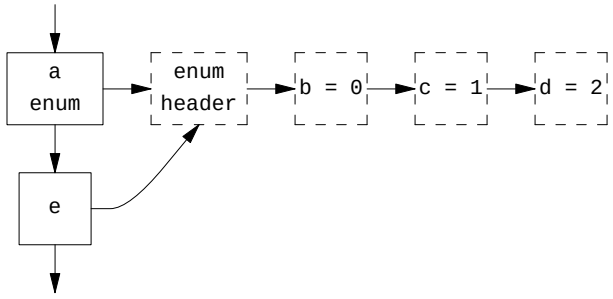
Next, here is an example of a declaration with more than one declarator; it shows how the type chains share a common end node. Again, there are no `type ref` nodes since the information which would normally be stored in such a node (const and volatile flags) is common to all the declared objects.

```
int a, *b, **c;
```



Lastly, the following diagram shows how an enumerated type is stored internally; it has a symbol table entry of its own so it can be found again in the future, similar to a structure definition.

```
enum a { b, c, d } e;
```



Type chains have a `reference count` in each node, which indicates how many other nodes are linked to this node. In the last example above, the reference count of the `enum header` node would be 2, and the reference count of the `b = 0` node would be 1. Type chains are deallocated by a function called freetype(), and this function stops deallocating when it hits a node whose reference count is greater than 1, decrementing its reference count instead. When a new link to a type node is created, its reference count must be incremented by one. When a block level in the symbol table is deleted, a freetype() is done on the types of all the entries deallocated, but usually, this does not delete the type chains since by that time, the parse tree will have many pointers to the chains and the reference counts will have been incremented accordingly.

The symbol table and type chains are built by the declaration processor, whose primary entry point is defined as follows:

```
process_declaration(node)
struct treenode *node;
```

This is called by the parser whenever it has assembled the parse tree for an entire declaration. The single argument is just the root node of the tree. The declaration processor builds symbol table entries and type chains for the declaration, and after it returns, the parse tree may be deallocated.

3.5. Type Checking

Type checking is the act of ensuring that C operators are applied only to types for which they are permitted, of ensuring that arguments to operators are promoted as required to types the operator can handle, and of determining what the type returned by the operator is. In tcc, typechecking is done at the same time as parse tree construction. Each parse tree node which corresponds to an expression which can return a value is linked to a type chain (typically all or part of the type chain of some symbol table entry) so that the type of the value can be instantly determined. Functions which build parse tree nodes according to this method are briefly discussed here. First, the functions which make tree leaf nodes:

```
struct treenode *make_var(id)
struct treenode *id;

struct treenode *eval_sizeof(type,node)
struct treenode *node;

struct treenode *make_const(num,uflag,lflag)
long num;
int uflag,lflag;

struct treenode *make_strconst(loc,len)
char *loc;
int len;
```

These functions respectively make the leaf nodes corresponding to identifiers, sizeof() operators (which always reduce to constant nodes of integer type), integer constant nodes (of unsigned and/or long type, as determined by the flags), and string constants. Each of these nodes is constructed with a type chain attached. For the make_var() function, if the identifier is a variable, the type chain is that of the variable. For the make_strconst() function, the type chain is 'pointer to signed 8-bit'. For the others, it is a basic type node. Next come the functions which construct non-leaf nodes, and do the actual type checking.

The make_cond() takes care of conditional operations, that is, operations which require or produce a "true/false" type of result. These include the C constructs `for`, `while`, `if`, the conditional expression `?:`, and the boolean operators `&&`, `||`, and `!`. Which of these operators is being handled is passed in the "type" argument. The function will construct the appropriate node and attach the appropriate type chain to it, but only after checking argument types and printing any necessary error messages or warnings.

```
struct treenode *make_cond(type,l1,l2,l3)
int type;
struct treenode *l1, *l2, *l3;
```

The make_arg() is used to ensure that a function argument has the appropriate type. At present, this involves only checking that the argument can be represented in a 32-bit machine word, and that it is not of structure, union, or array type.

```
struct treenode *make_arg(type,l1,l2)
int type;
struct treenode *l1,*l2;
```

The make_incdec() is used to type check the ++ and -- operators. It ensures that the operand is one to which the operation can legally be applied. If this is not the case, it deletes the operation and returns its argument node.

```
struct treenode *make_incdec(type,l1)
int type;
struct treenode *l1;
```

The next function makes a function call node. This involves making sure the first argument is indeed of type ‘‘function’’, or of ‘‘pointer to function’’ in which case it inserts a node to dereference the pointer automatically. Also, the task of this function is to automatically declare any undefined functions as external and returning integer.

```
struct treenode *make_functioncall(node1,node2)
struct treenode *node1,*node2;
```

Finally, a remaining function takes care of most of the other operators in the C language. This function is defined as follows:

```
struct treenode *make_inner(type,l1,l2,l3)
int type;
struct treenode *l1,*l2,*l3;
```

The basic task of this function, as of the others, is to generate a tree node, which it does by a call to make_plain(). But then it examines the operator for which the node is, and checks the types of arguments before proceeding.
The basic method is to produce a ‘‘cross product’’ of the argument types, and to handle each combination separately. To do this, a CROSS() macro has been defined, which takes two arguments and combines them into a unique result. Using this, type checking proceeds according to the following example, which might be for an ‘‘add’’ operation:

```
switch( CROSS(t1->type,t2->type,8) ) {
    case CROSS(TT_BASICTYPE,TT_ARRAYOF,8):
        ... convert array to pointer (by taking its address), then
        proceed with the following:
    case CROSS(TT_BASICTYPE,TT_POINTERTO,8):
        ... handle addition of an integer type to a pointer,
        generating a special “add” node which returns a pointer type.
    case CROSS(TT_ARRAYOF,TT_BASICTYPE,8):
        ... convert array to pointer (by taking its address), then
        proceed with the following:
    case CROSS(TT_POINTERTO,TT_BASICTYPE,8):
        ... handle addition of a pointer type to an integer as above.
    case CROSS(TT_BASICTYPE,TT_BASICTYPE,8):
        ... handle normal integer addition, generating an “add” node
        which returns an integer type.
}
```

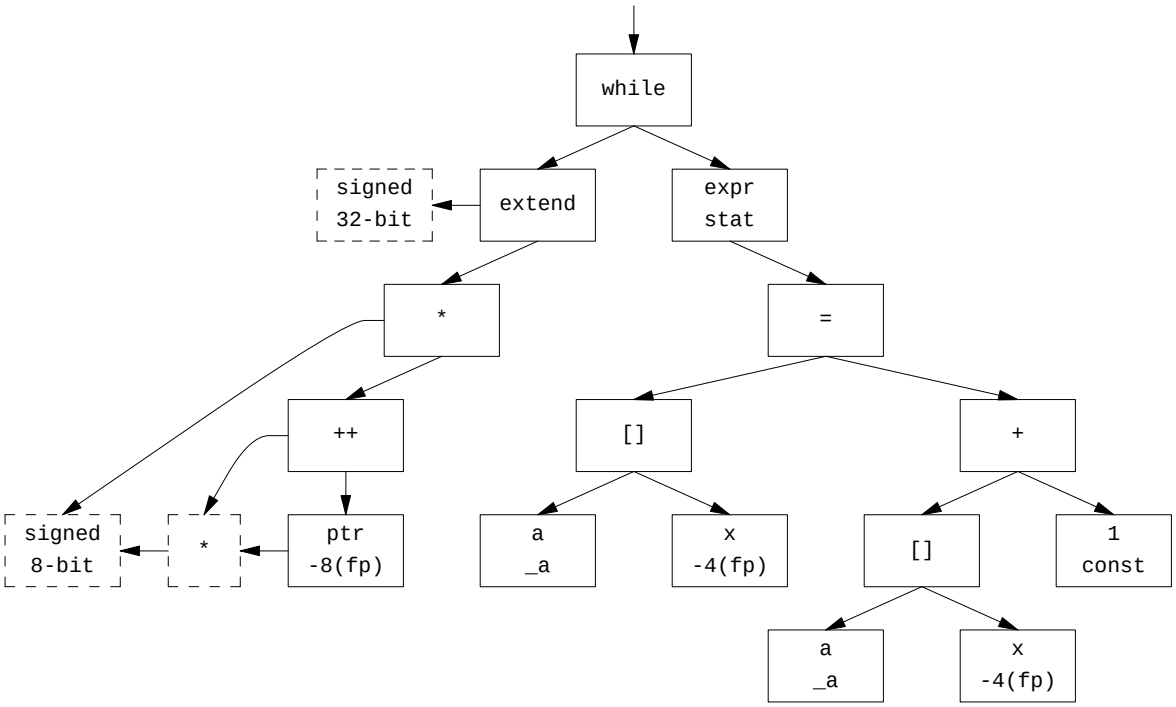
We conclude this section by illustrating what a complete parse tree, with type checking done, might conceptually look like. Suppose the following declarations were in effect:

```
int a[10];
char *ptr;
int x;
```

Where a is a global variable, and ptr and x are local variables on the stack. Then suppose the following code fragment were processed:

```
while(*ptr++) a[x] = a[x] + 1;
```

The parse tree for this is illustrated in the following picture. Note that this parse tree might be one statement in the compound statement of a function, and could be translated directly to code by a call to codegen() on its root, as discussed later. (in the diagram only the typenodes, shown as dashed boxes, for the conditional expression are shown)



The type chain for `ptr` was taken from the symbol table and is shown in dashed boxes at the lower left. The `++` operator accepts a pointer type, and returns the same type; therefore, the `type` field of its node points to the same type chain. The `*` operator accepts a pointer and returns the object to which the pointer points, in this case, the `signed 8-bit` node. Finally, an `integer promotion` is necessary before proceeding further, and as a result, the type checker has inserted an `extend` node to make the result a 32-bit type. This is presented as the conditional argument to the `while` node. Note that the code generator will ignore this particular `extend` node since, working its way down the tree rather than up, it realizes that it can just as well test a byte quantity directly, rather than extending to integer size and then performing an integer operation.

3.6. Constant Expression Evaluation

Constant expression evaluation is a necessary part of any compiler, since the values of certain expressions must be computable at compile time. In tcc, constant expression evaluation has been put into the functions which create parse tree nodes and do type checking for arithmetic operations (these are part of the

make_inner() function). These functions call another function named eval_const() early on. eval_const() examines the operation and the arguments, and if the arguments are all integer constant nodes (resulting from make_const() functions, eval_sizeof() functions, and other eliminated constant expressions), then it immediately evaluates the result and returns a constant node. When the type checking/node construction function sees this, it simply returns the constant node. Thus, at code generation time, to check for and evaluate a constant expression, one need only examine its tree node. If it is not an integer constant node, then something in the expression could not be evaluated and it was not constant.

A side benefit of the above scheme is that it automatically eliminates some constant subexpressions in non-constant expressions, without any additional effort required.

4. Code Generator Operation

The code generator recursively descends over the parse tree, transforming it into code. The most central function of the code generator is the translation of arithmetic expressions, and this function is discussed first. Next, the code generation technique for boolean expressions is explained, and finally, code generation at the statement level is demonstrated with examples of how it ties in with the other levels.

4.1. Expression Code Generation

4.1.1. The Expression Stack

Expression translation is done, conceptually, by a postfix traversal of the tree, which outputs operations in the order a stack machine could process them. Unfortunately, the target machine is not as simple as a stack machine. Thus the ''stack'' holding partially evaluated expressions is kept at compile time. We have called this the ''expression stack''. This is described in the following section.

The expression stack starts out empty. When an expression is evaluated, it holds the result, just as the stack would in a stack machine. The difference is that the expression stack holds only the information required to obtain the result, not the result itself. In tcc, this information takes the form of a VAX addressing mode. Each stack entry is described by a `type` field and two `value` fields, which we will denote as `x` and `y`. Then the expression stack can hold the following:

Name	DEC Notation	Unix Notation
ES_REG	Rx	rx
ES_REG+ES_DEF	(Rx)	(rx)
ES_CONST	#x	$x
ES_POSTINCR	(Rx)+	(rx)+
ES_POSTINCR+ES_DEF	@(Rx)+	*(rx)+
ES_PREDECR	-(Rx)	-(rx)
ES_DREGOFFSET	y(Rx)	y(rx)
ES_DREGOFFSET+ES_DEF	@y(Rx)	*y(rx)
ES_LABELADR	#_label+y	$_label+y
ES_STATICADR	#Lx+y	$Lx+y
ES_LABEL	_label+y	_label+y
ES_LABEL+ES_DEF	@_label+y	*_label+y
ES_STATIC	Lx+y	Lx+y
ES_STATIC+ES_DEF	@Lx+y	*Lx+y

The code generator begins at a leaf node of an expression by pushing onto the expression stack an addressing mode corresponding to the object there. The following table illustrates what each kind of addressing mode is pushed for various objects:

Type of object	Addressing Mode
constant expression evaluating to `x`	`#x`
global variable at label `_name`	`_name+0`
local variable at offset `y` from frame pointer	`y(R13)`
function parameter at offset `y` from argument pointer	`y(R12)`
quoted string (stored at Lx)	`Lx+0`
local static variable	`Lx+0`

Now as the code generator proceeds up the tree, it attempts to modify the addressing modes on the expression stack to represent the effect of the operator. Code is generated only when this is not possible to do. There are three major functions for manipulating the expression stack entries. They correspond to C operators as follows:

C operator	Function name
`.`	`addoffset()`
`*`	`indirect()`
`&`	`addressof()`

As an example, suppose the top entry on the expression stack is `_label+0`, and a dot operator is applied to it. It is found that the structure member named after the dot is at offset 16 into the structure. Thus the stack entry is simply transformed into `_label+16`. Or suppose that the expression stack contains `R0` and two `*` operators in succession are applied. The stack entry would become `@(R0)`. On the other hand, suppose that the stack entry already contains `@-16(R12)`, and another `*` operator is applied. There is no addressing mode which can represent this, so code must be generated. The `indirect()` function will now allocate a register, say `R0`, and generate this instruction:

```
MOVL    @-16(R12),R0
```

Then it will replace the top entry on the expression stack with `(R0)`. It should be noted that the code generator always attempts to have an ‘‘effective address’’ on the stack rather than a value, until this is no longer possible. This is to reserve the possibility of applying an `&` operator to the entry or to store something in it, which cannot be done once the entry is in a register. As an illustration, suppose a stack entry is `@(R0)` and three `*` operators are applied to it in succession. The code generator will generate this code:

```
MOVL    @(R0),R0            ; stack contains (R0)
MOVL    (R0),R0             ; still (R0)
MOVL    (R0),R0             ; still (R0)
```

And end up with `(R0)` on the stack. Note that it would have been more efficient to generate this code:

```
MOVL    @(R0),R0
MOVL    @(R0),R0
```

And end up with `R0` on the stack. However, the value would now be in a register rather than the address, and if the next thing to come along were an `&` or assignment operator, it could not be done. Therefore, the code generator, which as described so far cannot look ‘‘up’’ the tree to see if the address of an item will be required, has to sacrifice an occasional instruction to be on the safe side. Fortunately, constructs such as `***a` in C are very rare and thus this is not a big shortcoming.

4.1.2. The genexpr() Function

The heart of the expression translator in tcc is a function called `genexpr()`. It takes three arguments, as shown:

```
int genexpr(node,rp,maxsize)struct treenode *node;
struct es_entry *rp;
int maxsize;
```

The simplest invocation has the `rp` field set to zero and the `maxsize` field set to 4 (the size, in bytes, of an integer or pointer). Then the genexpr() simply translates the entire parse tree rooted at `node` (which must be the tree for an expression), and returns with an extra value on the expression stack referring to result. Suppose the expression `a+1` were to be translated, and `node` pointed to the tree node containing the `+` operation. Then genexpr would generate the following code:

```
ADDL3   _a,#1,R0
```

And return `R0` on the expression stack. Note that R0 is a value, not an effective address, but this is permissible and unavoidable since an expression like `a+1` can neither be taken the address of nor assigned to. It would, in fact, be a fatal internal error in the compiler if the type checker/semantic analyzer permitted such an operation at this point.

At this point, the `rp` argument to genexpr(), and the value it returns, should be discussed. These permit a simple lookahead mechanism which improves code generation. To illustrate why they are needed, suppose the expression `a = b + c` must be translated. Without the lookahead mechanism, the best the code generator could do is this:

```
ADDL3   _a,_b,R0
MOVL    R0,_c
```

It turns out that in the VAX, the result of an operation can often be stored in memory ‘‘for free’’ by just using a more complex destination field. That is, we wish to generate this instruction:

```
ADDL3   _a,_b,_c
```

This is accomplished using the ''advance knowledge'' the code generator has of the operations to be performed, since it processes the tree from the top down. Thus, when the = node is seen, the code generator already knows that the result of genexpr() for a+b will be stored in c. This is where the rp (for ''result pointer'') mechanism comes in. The code generator simply evaluates the left side of the = node first, resulting in an expression stack entry of _c. Now it calls the right side, with rp pointing to the stack entry of _c. This instructs the genexpr() function that if convenient, it can deposit the result directly there and not generate an expression stack entry at all. If this is possible, genexpr() will do it and return non-zero. If not possible, it will deposit the result on the stack as before and return zero. Here is another example where this is convenient. Suppose the result of a+b must be pushed on the stack as a function argument. Therefore, the caller to genexpr() constructs an expression stack entry of this form: -(R14) and points rp to it. Now genexpr() obligingly uses this as the destination, and generates this instruction:

```
ADDL3   _a,_b,-(R14)
```

Suppose on the other hand that genexpr() did not find it convenient to deposit the result directly. Assume the operation was ''push *a on the stack''. *a can be readily represented as an addressing mode, and so genexpr() will represent it on the expression stack as @_a without generating any code at all. It signals this by returning zero. Now the caller has to pop the entry off the stack and generate its own instruction, like this:

```
PUSHL   @_a
```

The only other argument to genexpr() is the maxsize parameter. This exists because not every result expected from genexpr() is 32 bits long. Some are shorter. maxsize serves two purposes. For one, it says that genexpr() need not bother evaluating the expression to any greater precision, and for the other, it says that when rp is set, genexpr() may only store a value exactly that big.

4.1.3. Extension of Short Values to Longer Ones

The definition of the C language states that short types must be ''integer promoted'' before practically anything can be done with them. The type checker takes care of this by inserting extend nodes into the parse tree where necessary. Suppose a and c are character variables, and b is an unsigned short variable, and the operation a = b + c is performed. The type checker would treat this like this:

```
a = promote(b) + promote(c)
```

Where promote(b) would be a node for ''zero fill upper 16 bits'' and promote(c) would be a node for ''sign extend upper 24 bits''. The addition would then be performed at the 32-bit (i.e. integer) level, as specified by the C standard, and the result would be truncated to fit into the 8-bit variable a. The code generator would see the = node first, and realize that the result need only be eight bits, and can be directly stored at location _a. It would thus call genexpr() with rp set to the entry for _a, and maxsize set to 1.

Genexpr() knows that for an addition operation to produce an eight bit result, only an eight bit add needs to be done. The same goes for subtract and multiply, but not for divide, for example. It thus calls genexpr() twice, each time with maxsize set to 1 but with rp set to 0. The next thing encountered on the way down is the extend nodes. These are processed by a function defined as shown:

```
int doextend(node,sign,sizenow,rp,maxsize)
int sign,sizenow,maxsize;
struct treenode *node;
struct es_entry *rp;
```

The function takes the `node`, `rp`, and `maxsize` arguments already discussed, plus the `sign` and `sizenow` arguments. `sign` simply specifies whether the extension is to be by sign extension or by zero filling, and `sizenow` specifies how big the value already is. So now the extend() function simply compares `sizenow` with `maxsize`, and if no extension is really required just calls genexpr() again. If an extension is required it only extends as far as necessary, for example, to 16 bits rather than 32. And finally, it can deposit results directly using the caller's mechanism. Thus in the previous example, neither of the `extend` nodes cause any code at all to be generated, and the whole statement `a = b + c` translates to this:

```
ADDB3    _b,_c,_a
```

Consider another example of ''push signed byte quantity a on the stack''. doextend() has enough information to generate just this instruction:

```
CVTBL    _a,-(R14)
```

It can be seen that the `rp` mechanism has allowed integrating the `push` operation with the ''extend'' operation. Finally, consider the example ''copy unsigned byte quantity a to word quantity b'' doextend() generates this code for this:

```
MOVZBW   _a,_b
```

4.1.4. Register Allocation and the Stack

The VAX has only a limited number of registers to work with, and occasionally there arise situations where not all temporary values can be held in registers at once. The first such situation is when the ''depth'' of an expression becomes too great, i.e. there are more temporary values than available registers. The second situation is when a function is called. Functions can be called from various contexts, and so they do not know what state of the register set is in and simply overwrite registers at will (this is the method used by the system compiler with which tcc is object module compatible).

Therefore, a method is needed to keep temporary values somewhere other than in registers. This is provided by the `(R14)+` addressing mode, which pops a value off the stack, and the `@(R14)+` addressing mode, which pops an address off the stack and accesses the value at that address. When a register is needed and none is available, the register allocator will traverse the expression stack from the bottom, and turn the first addressing mode which it finds which uses a temporary register into a stack one, as shown in the following table:

Old Mode	Code generated	New Mode
`Rx`	`PUSHL Rx,`	`(R14)+`
`(Rx)`	`PUSHL Rx,`	`@(R14)+`
`y(Rx)`	`PUSHAB y(Rx)`	`@(R14)+`
`@y(Rx)`	`PUSHL y(Rx)`	`@(R14)+`

Since the lowest entry in the expression stack which can surrender a register is thus transformed, the flushed values end up on the system stack in just the order in which they will be needed, and so the ''pop'' addressing modes can be used just like the original ones.

Some precautions are in order here. First, at no point may two operands in the same instruction use a ''pop'' addressing mode unless it is absolutely certain that the VAX CPU will ''pop'' the values in the proper order. Second, if the value to be popped is less than a longword, then building the ''pop'' right into an instruction may cause only part of the pushed longword to be popped. Both these situations are prevented by careful coding of the code generator, with explicit popping of stack contents back into registers

before use where unavoidable.

4.1.5. Expression Translation Example

Having discussed the main aspects of the expression translator's operation, we will illustrate how the concepts work together with a simple expression translation example. The expression to be translated (in context) is:

```
short a;
int *b;
int c[20];

main()
{
    a = *(b+1) + func(a) + c[5];
}
```

The actual tcc output for the expression follows:

```
addl3   $4,_b,r0
pushl   r0
cvtwl   _a,-(r14)
calls   $1,_func
addw2   *(r14)+,r0
addw3   _c+20,r0,_a
```

The first line computes the address of `*(b+1)` and loads it into a register. The second line flushes this register to the stack since calling a function will overwrite it. The next line extends a to a longword and pushes it on the stack, then the function is called. Finally, the two add operations are performed using the addressing modes which have accumulated on the expression stack: `*(r14)+` from the `*(b+1)`, and `_c+20` from the `c[5]`. The result is immediately dumped in the destination location by the last instruction thanks to the `rp` mechanism, and the additions are performed at word precision thanks to the `maxsize` mechanism. Note that the code isn't perfect: The code generator could have rearranged the expression to eliminate at least the `pushl r0` instruction, but this compiler is not optimizing and the local optimizations it does make can only go so far.

4.2. Boolean Expression Code Generation

The C standard specifies that the evaluation of boolean expressions must be ''short-circuited'' that is, evaluation of the expression must proceed from left to right and stop as soon as the final result is known. This section explains how this is done in tcc.

4.2.1. The gen_logicexpr() Function

The function is the second of the three most important functions in the code generator (the third is described in the next section). It is declared as follows:

```
gen_logicexpr(node,ltrue,lfalse,lnext)
struct treenode *node;
int ltrue,lfalse,lnext;
```

The function can be called in two ways. From the statement level, it is called with three label arguments, representing the label to jump to if the expression is true, the one to jump to if the expression is false, and the number of any label which will be generated immediately after the function returns (or zero if none). From the expression level, it is called with the three label numbers all set to zero, in which case the generated code will return success or failure as the value 1 or 0, and gen_logicexpr() will leave information on

how to get this value on the expression stack.

4.2.2. Handling of &&, ||, and ! Operators

The handling of the boolean operators in gen_logicexpr() is extremely simple. The function simply generates labels and calls itself, as appropriate. For example, with ltrue, lfalse, and lnext all set as shown in the previous section, the handling of the `!` operator is simply this:

```
gen_logicexpr(node->link1,lfalse,ltrue,lnext);
```

The true and false labels have simply been reversed, reversing the result of whatever the argument of `!` returns. The handling of the `||` operator is only slightly more complex:

```
lab = labelcnt++;
gen_logicexpr(node->link1,ltrue,lab,lab);
genlabel(lab);
gen_logicexpr(node->link2,ltrue,lfalse,lnext);
```

Here, the first line generates a new label number. The next line processes the left argument of the `||`, with label numbers set so that a false return causes the next expression to be evaluated, and a true return causes the entire expression to become true. The next line generates the label, which is duly indicated in the `lnext` field of the gen_logicexpr() call. Finally, the right side of the `||` operator is evaluated, with the label fields set so that the entire expression succeeds or fails depending on the result. Since we are about to return, lnext can be set to whatever was given for the whole expression. It should be clear at this point how the && operator would work, and the code for it is not shown.

4.2.3. Handling of Comparison Operators

The six comparison operators are all conceptually handled in the same way. Both arguments are evaluated and deposited on the expression stack by calls to genexpr(), then the appropriate comparison instruction is generated, and the arguments are popped off the stack.

The first detail to point out is the method used to choose the size of the comparison necessary. This involves looking down both subtrees, past any `extend` nodes resulting from the inevitable integer promotion of the arguments, to see how big the arguments really are. Then a compare instruction of the size of the bigger operand is chosen, and genexpr() is called with `maxsize` set to that size. Also, when determining whether comparison instructions should be signed or unsigned, the signed/unsigned property of the larger of the two operands is used, unless they are equal in size in which case the comparison is unsigned if either of the two is unsigned.

For example, if the unsigned short integer `a` and the signed character `b` are to be compared, then `maxsize` is 2 and the comparison will be unsigned.

4.2.4. Efficient Generation of Conditional Branches

Nominally, each conditional branch generated in a logic expression would take this form:

- Evaluate condition
- Branch to ltrue if true
- Branch to lfalse if false

However, it is often the case that either ltrue or lfalse immediately follow the above, and thus one of the branches may be unnecessary. It is clear that to reliably eliminate the second branch in this case, the code generator must have access to both a ‘‘branch if true’’ and a ‘‘branch if false’’ version of each branch. Then, if ltrue immediately follows, this code can be generated:

- Evaluate condition
- Branch to lfalse if false

And if lfalse immediately follows, this code can be generated:

- Evaluate condition
- Branch to ltrue if true

Finally, if neither ltrue nor lfalse follow immediately, then the code shown earlier must be generated, except that the second branch can be unconditional. All this work is done by the gencjmp() function, which is defined as follows:

```
gencjmp(optrue,opfalse,ltrue,lfalse,lnext)
char *optrue,*opfalse;
int ltrue,lfalse,lnext;
```

Its arguments are, respectively, the branch instruction which will branch if the condition is true, the one which will branch if the condition is false, and the three label numbers discussed earlier. For example, the following call to gencjmp() generates code for ‘‘jump to l1 if the zero flag is set, l2 if it is not set, with l3 following next’’:

```
gencjmp("jeql","jneq",l1,l2,l3);
```

The function will generate one of the following forms of code, depending on the relationship of l1, l2, and l3:

```
    jeql    l1          jneq    l2          jeql    l1
l2: ...             l1: ...                 jbr     l2
                                            ...
```

4.2.5. Returning Boolean Values

When gen_logicexpr() is first invoked, it checks if ltrue, lfalse, and lnext are provided (i.e. non-zero). When this is the case, it has been called either from itself, or from the statement translator, and need only transfer control to labels already provided. But if they are zero, it has been called from the expression translator, which wants the result given as a value on the expression stack.

In this case, gen_logicexpr() allocates its own labels at the start, and when the expression is finished, generates the labels along with code to load 0 or 1 into a register. Then it pushes that register on the expression stack and thus returns a value to the caller. The `rp` and `maxsize` mechanism is not provided here, so that the value is always returned in a register.

4.2.6. Boolean Expression Example

For an example of boolean expression translation, we will consider the following code:

```
char a,b;
unsigned short c;
int d,e;

main()
{
    e = !(a<c) || b && (c!=d);
}
```

The actual tcc output for the expression above is shown below:

```
        cvtbw   _a,r0
        cmpw    r0,_c
        jgequ   L1
L3:     tstb    _b
        jeql    L2
L4:     movzwl  _c,r0
        cmpl    r0,_d
        jeql    L2
L1:     movl    $1,r0
        brb     L5
L2:     clrl    r0
L5:     movl    r0,_e
```

First, `a<c` is processed. Since `c` is longer than `a`, `a` is extended to the size of `c`, and since `c` is unsigned, an unsigned branch is used. Due to the `!` and `||` operators, the expression immediately succeeds if `a >= c`, and the conditional branch generated reflects exactly this, branching to the point where 1 is returned from the expression. Next, `b` is tested, and if it is zero, then the expression immediately fails, returning zero. Finally, `c` and `d` are brought to a common size and compared, and if they are equal, then the expression again fails. Finally, control falls through to the point where 1 is returned. It is seen how the problem of branches branching around other branches has been entirely avoided.

4.3. Statement Code Generation

The previous section explained the most difficult aspect of the code generator, namely, how translation works at the expression level. This section describes how translation works at the statement level.

4.3.1. The codegen() Function

At the highest level, the code generator is visible as a single function (the most important one), which is defined as follows:

```
codegen(node,lab1,lab2)
struct treenode *node;
int lab1,lab2;
```

The function takes as its argument a tree node which must be the root of a parse tree representing a complete statement. The other two arguments are label numbers, indicating respectively the place where a `break` statement and a `continue` statement should transfer control. These arguments are zero when the respective statement is not valid, i.e. when there is no enclosing statement providing a place for it to jump to. The parser initially calls codegen like this:

```
codegen(node,0,0);
```

Where `node` is the root of the entire compound statement making up the body of a function.

The codegen() function handles flow-of-control constructs, compound statements, and individual ''expression statements'' For example, for a compound statement, it traverses the linked list of statements inside, calling itself for each. For an expression statement, it calls genexpr(), then discards the result genexpr() leaves on the expression stack by simply popping it off. Actually, it calls a special front end to genexpr() when the result is not required. This front end, called eval_and_toss(), handles certain expressions slightly better, knowing that their results will not be needed.

4.3.2. Control Flow Construct Compilation

The fact that the codegen() function is recursive, and that it has the gen_logicexpr() function at its disposal to generate conditional branches, makes the handling of control flow constructs very simple. We will show as an example the entire part of codegen() responsible for `while` loops:

```
l1 = labelcnt++;
l2 = labelcnt++;
l3 = labelcnt++;
genlabel(l1);
gen_logicexpr(node->link2,l2,l3,l2);
genlabel(l2);
codegen(node->link3,l3,l1);
en.type = ES_STATIC;
en.u.val1 = l1;
en.val2 = 0;
geninst("jbr",1,&en);
genlabel(l3);
```

The first three lines simply reserve three label numbers, which are respectively: top of loop, ''true'' label for the conditional expression, and end of loop. Then the first label is generated, and the conditional expression is translated so that control transfers to the next statement if it is true, and to the end of the loop if it is false. Then the ''true'' label is generated, and the body of the loop is compiled via a recursive call to codegen(). Finally, a branch to the top of the loop is generated (four lines), and the end label is emitted. It is seen how the proper label numbers are passed down in the recursive call to codegen(), directing `break` and `continue` statements to their places.

The code to compile the other control flow constructs is equally simple, and so it will not be specifically discussed.

4.3.3. Switch Statement Implementation

A great deal of care was taken to have an efficient implementation of the C language `switch` statement. This statement is best implemented as a dispatch table, with an entry corresponding to each of the possible values of the selector operand, with all those entries for which no case was provided branching to the default location. Unfortunately, not all switch statements have case values which are closely clustered together and permit this type of thing. For this type, tcc builds an efficient compare-and-branch dispatcher, which will be described.

Compilation of switch statements begins by a traversal of the parse tree below the switch node, extending down to the individual statement level but not going into the compound statements attached to switch nodes. A label number is assigned to each case, and the case values and label numbers are collected in a dynamically allocated array (therefore, there is no limit on the number of cases). Next, the maximum and minimum case value in the array are found, the array is sorted by case value, and duplicates are eliminated. Then it is time to decide which dispatch implementation will be used. The criterion is to use the dispatch table if there are more than two cases and if the dispatch table will have at least one entry out of 10 going to a case (with the others going to the default). If both are not met, then the compare-and-branch dispatcher is used.

4.3.3.1. Dispatch Table Implementation

The dispatch table implementation uses a VAX `case` instruction. The instruction is not described here, but an example is shown for the following code fragment:

```
switch(a) {
    case 1:  < statement at L1 >
    case 2:  < statement at L2 >
    case 5:  < statement at L3 >
    default: < statement at L4 >
}
```

The tcc output corresponding to the above skeleton would be:

```
        subl2   $4,sp
        casel   _a,$1,$4
L5:     .word   L1-L5
        .word   L2-L5
        .word   L4-L5
        .word   L4-L5
        .word   L3-L5
        jbr     L4
```

It is assumed that the reader is familiar with the VAX `case` instruction and thus this will not be discussed further.

4.3.3.2. Compare-and-branch Implementation

When each case value has to be explicitly tested against, the dispatch time for the switch statement becomes painfully dependent on the number of different cases. For a straightforward compare/branch on each case value, the dispatch time would obviously be O(n). Fortunately, there is a simple method, requiring only a little more code, with which this dispatch time can be reduced to $O(\log_2(n))$, and this was used for tcc. The method is to generate a binary tree-shaped dispatch system, in which the test against each case value eliminates half the remaining case values. A three-way branch is required after each compare: either go to the code for that case, or go check lower values, or go check higher values. The routine which does this is a straightforward binary search of the case array, except that instead of searching it generates code which traces its steps. Its output will be shown for the following skeleton:

```
switch(a) {
    case 1:         < statement at L1 >
    case 4:         < statement at L2 >
    case 16:        < statement at L3 >
    case 64:        < statement at L4 >
    case 256:       < statement at L5 >
    case 1024:      < statement at L6 >
    case 4096:      < statement at L7 >
    case 16384:     < statement at L8 >
    default:        < statement at L9 >
}
```

The tcc output for this skeleton would be as shown below (in two columns to save space):

```
        movl    _a,r0               jbr     L9
        cmpl    r0,$64      L11:    cmpl    r0,$1024
        jeql    L4                  jeql    L6
        jgtr    L11                 jgtr    L12
        cmpl    r0,$4               cmpl    r0,$256
        jeql    L2                  jeql    L5
        jgtr    L10                 jbr     L9
        cmpl    r0,$1       L12:    cmpl    r0,$4096
        jeql    L1                  jeql    L7
        jbr     L9                  cmpl    r0,$16384
L10:    cmpl    r0,$16              jeql    L8
        jeql    L3                  jbr     L9
```

As can be seen (by tracing through the code for various values of a), no more than three comparisons are ever necessary to reach the proper case or the default.

4.4. Code Generator Summary

There are many other functions in the code generator, but there is no space in this document to describe them all. It is hoped that the description given in this section has given the reader the knowledge necessary to understand the source code of the code generator by describing major highlights throughout, and the philosophy used in the design of the individual routines. This is a hard-coded as opposed to a table driven code generator primarily because it was thought that the design for a table driven scheme, the implementation of the table interpreters, and the writing of the tables would be far more time consuming than writing the somewhat less than 2000 lines of C which the code generator presently consists of has been. Indeed, it was possible to debug the code generator, as it was written, a small piece at a time. Less than 30 lines of code had to be changed in further debugging once the whole compiler got off the ground and large-scale tests of more than 10-20 lines could finally be compiled.

5. Compiler Limitations

This section describes limitations in the design, and encountered during the construction of the compiler.

5.1. Design Limitations

The following points describe areas in the definition of the C language that were not included in the design of the compiler.

1. Floating point (data types float, double) is not supported by the compiler.
2. Aggregate objects (structures, unions, arrays) cannot be passed to functions as parameters or returned from functions. All data types passed to functions or returned from functions must fit in 32 bits (therefore, a long integer or a pointer is the biggest object which can be passed).
3. ANSI style function definitions are not supported. ANSI style function definitions have their entire parameter declarations in the parentheses of the function declarator, and can be predeclared for argument type checking and promotion. The compiler only supports "old style" functions, in which the function can be predeclared as to return value only, and in which all arguments are extended to sizeof(int) and passed in equal sized slots on the stack.
4. Bit fields in structure definitions are not supported.
5. The keywords `const` and `volatile` are not supported. At present, they are parsed by the declaration processor, and stored in the internal symbol table data structures. But the type checker/semantic analyzer, whose job it would be to prevent writing to `const` objects, does not do so at present. The `volatile` keyword is unneeded at any rate, as the code generator does not optimize sufficiently to make it necessary.
6. Register variables are not supported. Any `register` keywords are treated exactly like `auto` keywords.

5.2. Implementation Limitations

The following points describe limitations that came into being during the construction of the compiler.

1. Due to the design of the ANSI C grammar used for the parser, it is not possible to fully support the scope rules of a user defined type. This is because the grammar depends on the lexical analyzer to distinguish between type identifiers and regular identifiers by consulting the symbol table, and return one of two tokens appropriately. The grammar further lacks the facilities to parse a declaration in which type identifiers are present as anything but declaration specifiers. This makes it impossible to "hide" a defined type in an inner scope, as there is no way to process the declaration which would hide it. As an example, the following program will fail to compile, though it is a valid ANSI C program.

```
typedef int INT;
INT a;

main()
{
   int INT = 1;
}
```

2. The code generator does not support conditional expressions returning a value which cannot be stored in a register. A register will be allocated to store the result, then both alternative expressions will be translated and forced to each return their result in this common register. A control structure similar to that of an IF/ELSE is built to select between the expressions. Since structure/union types can not normally be held in a register, and `void` types have no value to hold in a register, the type checker prevents such types from reaching the code generator. The following code fragments illustrate the problem constructs:

```
struct { ... } a,b,c;
void func();
int d,e;
...
a = d ? b : c;
d ? (void) e++ : func();
```

This does not impose an undue limitation on the programmer, however, since the conditional expressions could easily be rewritten like this, for example:

```
a = *(d ? &b : &c);
if(d) e++; else func();
```

3. Unsigned long versions of the modulo (%) and right shift (>>) operations are not supported. This is because the VAX does not have unsigned division or unsigned right shift operations in its instruction set. The existing C compilers perform unsigned long right shifting with a ''bit field extract'' instruction, and unsigned long modulo using a library function to do the unsigned division. It was arbitrarily decided not to bother supporting these in the code generator for tcc, especially as unsigned types were an optional part of the project. Unsigned division is fully supported, however, including those cases where the library function must be called.

4. For array dimensions, the internal representations of `[]` and `[0]` are the same. It appears that the existing C compilers make a distinction here, allowing arrays of size zero to be allocated as global variables or structure members (no storage is reserved for them in either case), but not allowing the same thing to be done with incompletely defined arrays. tcc considers all arrays of size zero to be incompletely defined, and does not allow such arrays to be used in a context where their size must be known.

5. The constant expression eliminator is guaranteed to eliminate expressions which are entirely constant, but not guaranteed to spot and reduce all constant subexpressions. This is due to the order in which the parser recognizes the expressions. As an extreme case, the expression

```
a + 1 + 2 + 3
```

is parsed in this order:

```
(((a + 1) + 2) + 3)
```

due to the left-to-right associativity of the + operator. It thus becomes non-constant right away and code is generated to perform all four additions. It would take major work to improve on this because the commutativity of + and other operators would have to be taken into account and the parse tree rearranged before things as the above example could be simplified. In the mean time, the programmer writing tcc can easily help the compiler along by separating constant subexpressions in parentheses; then they are always seen and reduced. Thus the previous example would become

```
a + (1 + 2 + 3)
```

6. It is not possible to define an enumerated type and initialize a variable to one of the enumerators on the same line. This means that the following will not work:

```
enum a { one, two, three } b = one;
```

The reason for this is that an entire declaration is parsed by the parser and then handed to the declaration processor as a unit. Thus the left side of the declaration has not been processed by the time the right side is parsed, and thus "one" is not yet known as an enumerator. Thus an error will be generated for undefined identifier "one". This could be fixed by splitting out the processing of `struct`, `union`, and `enum` type specifiers to be performed immediately after such a type specifier has been parsed, then passing a pointer to the resulting symbol table entry to the declaration processor. It is estimated that perhaps 50 lines of code in the area of `handle_struct`, `handle_enum`, and `process_declaration_specs` would have to be changed to do this.

7. The construct `"string1" "string2"` in ANSI C must be concatenated to form `"string1string2"`. Tcc does not do this. It could be added by modifying the string gathering function in the lexical analyzer to skip white space after it sees the closing quote, and continue gathering if the first thing after the white space is another quote. It is estimated that perhaps 30 lines of code might be required. Tcc does support the old style convention of a backslash/newline combination being ignored in a string, allowing very long string constants to be built that way.

8. The constant expressions for initializers are restricted. Integers may only be initialized to integer constant expressions. Pointers may be initialized to one of the following:

   - an integer constant expression cast to the pointer type
   - an integer constant expression evaluating to zero
   - the address of an identifier, taken with a single &
   - the name of a function or array (in which case the & is implied)
   - a quoted string (for character pointers)

These have been chosen as a best compromise between ease of translation and usefulness. They provide for dispatch tables to functions, and tables of ASCII strings, for example. But it is illegal to have expressions such as the following (valid in ANSI C) in initializers compiled by tcc:

```
&id1 - &id2 (to initialize an integer)
(int) &id1
&id1 + 1    (to initialize a pointer)
&array[5]
```

All other aspects of initialization, including initialization of character arrays using quoted strings, and the rules for grouping of initializer expressions using `{` and `}`, are fully supported.

9. Enumeration types have been supported as a distinct type all the way through type checking. This makes it possible to issue a warning when a value whose type is not the proper enumerated on is assigned to an enumerated variable. Unfortunately, it necessitates many implicit conversions between `enum` and `int` in the type checker, so that `enum + enum` returns `int`, for example. It is likely that this is not completely handled and that situations can be found where integers work and enumerators will not. At this writing, for example, it is impossible to add an enumerated variable to an array acting as a pointer.

10. The declaration processor does not check for duplicate member names in a structure or union, duplicate enumerators in an enumeration, or a conflict between an enumerator and some other identifier. This is harmless: two structure members or enumerators with the same name cause the second one to be invisible, and a conflict between an enumerator and another identifier always causes that other identifier to be seen first. But warnings could be generated in such cases, with the addition of another 30 lines of code or so to the declaration processor and a small run-time performance penalty.

11. Fixed-size arrays are used in various places, and a couple of these are not checked for overrun. It is possible to crash the compiler with a string constant longer than 1000 characters, or a file name longer than 80 characters. These limitations can be increased by editing ‘‘scan.l’’ and ‘‘main.c’’ and recompiling. An additional 20 lines of code or so added to the string gathering function and the mainline would remove this potential problem.

12. The code generator does not use the ‘‘index’’ mode of the VAX instruction set at all. Supporting this would have necessitated addressing modes which tie up two registers (one for the base addressing mode and one for the index) and this was judged too complex. In addition, the index mode generates an offset which is dependent on the size of the operation, which is not known at the time the addressing mode is built, so structural changes to the code generator would be necessary as well.

13. It is not presently possible to pass options to the C preprocessor when it is invoked automatically by tcc. If such options are required, the preprocessor must be invoked explicitly like this:

```
/lib/cpp {options} file.c | tcc -n -o file.s {other options}
```

    This is judged sufficient, since preprocessor options are rarely used outside a Makefile, in which the syntax above would be acceptable. In any case, a 20-30 line modification to the mainline would fix this problem.

14. The error messages from the compiler are clumsy at best. The error messages generated at parse time attempt to point out the error location in the source code; however, due to the order in which the code is parsed the errors are only seen at the close of a syntactical construct. For example, in a `while` loop, where the argument to `while` has an invalid type, the error will not be reported until the end of the loop body, at which point the `while` grammar rule is finally reduced and the argument type checked. The result is a pointer to a totally irrelevant position from one to many lines below the actual error. It is likely that eliminating attempts to point out the error location for all but syntax errors would be a good approach, using instead the line numbers built into the parse tree nodes like is done in the later stages of the compile process. A feature which points out at which token the error occurred, such as found in the system C compiler, would also be worthwhile. And finally, the error productions in the C grammar are still insufficient to recover cleanly from syntax errors. There are cases where a single syntax error causes the compiler to spew out ten error messages or more.

6. Compiler Testing and Validation

6.1. Self Compilation

The main test of tcc was to see if it could compile itself. This was accomplished. The self-compiled version of tcc was tested by using it to compile itself again, producing a purely native version which the system compiler had no part in producing. The source files from this first compilation were saved, and the compiler was recompiled yet again. The source files were compared, and found to be identical. **All** further testing of tcc was performed with a second-or-greater generation self-compiled tcc, i.e. one which had been compiled using the self-compiled version of itself. Several small debugging updates were also made by incrementally recompiling tcc with itself. We believe that this is enough evidence to claim that tcc is self-hosting.

The Makefile which is supplied in the code listings with this report is the one used to compile tcc with itself.

6.2. Supplied Test Cases

Each of the supplied test programs was compiled with tcc. The sources and assembly language outputs are given in the printout section. A typescript of the compilation and execution of the tests is also provided. Briefly, all tests compiled and executed correctly except for the following:

externs.c − did not work because floating point numbers are not implemented in tcc. A modified version, called ‘‘externs-modified.c’’ with the floating point types replaced by integral types, worked

correctly.

pointer.c − compiled correctly but did not run, since absolute addresses 0x20000 - 0x20100 are not readable by default on the VAX Unix system. A modified version, called ‘‘pointer-modified.c’’ was made which reads 0x100 bytes starting at symbol ‘‘main’’ and this worked correctly.

struct3.c − used a structure before it was defined, and thus did not compile. The modified version (struct2.c) which does things in the proper order worked correctly.

6.3. The CPR Utility

The cpr utility had one line in it which tcc could not handle. It was the following:

```
enum langs {NONE, AUTO, C, FORTRAN, ICON, LISP} Language = AUTO;
```

The reason was that tcc has the documented limitation of not being able to define an enumeration and use enumerators from it in initializations in the same declaration. When the line was changed to read:

```
enum langs {NONE, AUTO, C, FORTRAN, ICON, LISP} Language;
```

With an explicit initialization early in the mainline like this:

```
Language = AUTO;
```

CPR was compiled with no further warnings or errors of any kind, and the resulting version of CPR was used to print all listings included with this report.

6.4. Obfuscated C Contest Entries

A number of winning entries from past years of the ‘‘International Obfuscated C Contest’’ were run through tcc. As expected, these stressed the compiler more than most ‘‘normal’’ programs. Some, which primarily abused the C preprocessor, were not bothered with. Others caused huge cascades of error messages from tcc whose causes were not determined. One caused tcc to produce incorrect code which core dumped when run. Again, due to the nature of the programs, this was not further investigated. But eight programs did compile without modification and ran correctly and printouts of these are included in the test section of the listings. There are two ‘‘Hello World’’ programs, one ASCII to Morse Code converter, one brutal abuse of the ‘‘goto’’ statement, a program which plays Othello, a program which bubble-sorts the characters of the strings in its argument vector, a program which scrolls a message across the display, and a program which converts Arabic to Roman numerals.

6.5. Miscellaneous

6.5.1. Tetris

A particularly nice Obfuscated C Contest entry, which implements a complete and playable game of Tetris, was made to work with tcc courtesy of Robert Parker, a 2A Math student who has been instrumental in the stress testing of tcc. As seen in the comments in the source code (included with the listings), only two minor changes had to be made to get the program to compile, and one of those corrected a violation of the ANSI standard.

6.5.2. Course Assignments

Two programs from computer engineering course assignments, totaling perhaps 1500 lines of code, not counting comments, were compiled with tcc. As expected they compiled flawlessly. The programs, which are a cache simulator and a small database manager, are available online with tcc and the rest of the tests, but have not been printed.

6.5.3. Bob Parker's Specials

Three test cases by Bob Parker (pr1.c, pr3.c, and swamp.c) may be found in the listings. These were verified to give the same results when compiled with tcc as they do when compiled with the system compiler. We think they stress the compiler a little more than the supplied test cases do.

6.5.4. Public Domain Utilities

A few public domain Unix utilities were compiled with tcc and verified to work, again by Bob Parker. No modifications to the sources were necessary. At present, the list is the following:

- stat (a directory lister)
- tab (a simple indent program)
- weekday (a program to compute information about dates)
- which (a program which shows what commands are aliased to)

The sources and executables as compiled with tcc are available online.

6.5.5. Expression Stress Test

To test the the flushing of registers to the stack in the tcc code generator, an extremely complex and ''deeply nested'' expression was written. It was verified that flushing registers to the stack was required to keep all the temporaries around, and then in addition a number of function calls were inserted into the expression to cause further register flushes. The expression was tested with a thousand inputs of six random numbers each, generated with a program also compiled with tcc. The results were compared to those obtained when the expression was compiled with the system compiler, and found to be identical. This test can be found in source files ''exprtest4.c'' and ''genrand.c''.

7. Compiler Performance

To evaluate the quality of code generation of the compiler, it was compared to the system C compiler. The system C compiler was ''handicapped'' by not allowing it to use register variables, reflecting the fact that these were not implemented in tcc. This was done by using ''-Dregister='' as one of the compiler options. Then the entire source code for tcc was compiled with both the system compiler and with tcc itself. The results (stripped executables) were as follows:

```
tcc (self compiled):        95232 bytes
tcc (system compiled):      93184 bytes
```

It is seen that for this particular case, tcc output was 2.2% larger than system compiler output. It is recognized, of course, that one can get much better code by using register variables and the ''-O'' option with the system compiler, and better code yet by using the GNU compiler, but the goal here was to test the performance of a 3-month hack compiler against the simplest ''real'' compiler available.

Next, the self compiled and system compiled versions of tcc were both used to recompile one of the tcc source files (genexpr.c) and the runs were timed. The following results were obtained:

```
tcc (self):         52.5u 3.2s 3:30 26% 126+369k 19+17io 7pf+0w
tcc (system):       51.3u 2.8s 1:41 53% 125+367k 28+19io 15pf+0w
```

While it is recognized that some of that time is spent in the C preprocessor whose execution time is the same in both cases, it is not unreasonable to assume that most of the time is spent in tcc itself, and thus that the system compiler produces code which is only marginally faster (approx. 3-5%) than that of tcc. Again, the performance increase of the system compiler code would likely be signifi cant if register variables and optimization were allowed. Just for a test, genexpr.c was again compiled, but with the system compiler itself (the ‘‘-S’’ option was used to prevent the assembly phase). The times were as follows:

```
system compiler:   35.1u 1.9s 1:01 59% 151+329k 31+28io 5pf+0w
```

It is seen that the system compiler compiles about 46% faster than tcc does, though this would decrease if tcc were recompiled with register variables (there are register declarations in the tcc source code).